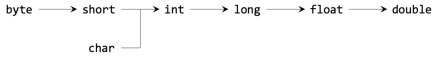
CS2030S Midterm

AY24/25 sem 2

github.com/mendax1234

Intro to OOP

1. **(Constructor)**: If your class includes a **constructor with parameters**, you are required to **provide arguments** when creating an object using that constructor.
2. **(Target)**: `object.method()`, `object` is the **target**.
3. **(Initialization)**: Any *reference* variable that is not initialized will have *null*. Any *primitive* type variable will have either 0 or false (boolean).
4. **(Java)**: **Java** is a **statically typed** and **strongly typed** language.
   - **Statically typed**: the variable can only hold values of the **declared type**. (Any subtype of the declared type is allowed).
   - **Strongly typed**: Java compiler enforces a **stricter type checking rule to ensure type safety**. These rules include:
     - **Valid subtype relationship when assigning one variable to another**: **Implicit narrowing conversion** is not allowed and **Type casting that makes no sense** is not allowed. (No subtype relationship).
   - **Floating point number** is treated as `double` by default. **Integer Literal** is treated as `int` by default.
   - **Local variables won't be initialized** and accessing **uninitialized local variables** will generate **compile error**!
   - **Subtype Relationship**
     - **Primitive type**:



     - **Wrapper class**: `Number` is the **superclass** of `Integer, Double, BigInteger, Long` etc.
   - **We cannot instantiate one object twice**, e.g. calling `Circle c1 = new Circle();` twice will generate a **compile error**!
   - We **cannot change the CTT of a variable**. So, if we have declared `Circle c`, we **cannot use** `String c` anymore in the same segment.
   - Java is a **strongly typed language**, but it allows **widening type conversion** and will do this **automatically without explicit casting**.
   - In Java, two types without a subtype relationship **cannot** be casted.
   - **Nested method calling**: In Java, the nested method call is executed from **left to right**. e.g. `Box.of("string").map(new StringLength()).map(new AddOne());`, the left `.map` will be executed first.
   - **method return**: Suppose the return type of a method is **T**, inside this method, you can actually return the **subtype** of **T**.
   - **Reflexivity of subtype**: If `A<:B, B<:C`, then `A<:C`. Pay attention to the reflexivity!!! It may generate all the correct answers!
5. **Information Hiding**:
   - **fields** should be declared as `private`
   - **methods** should be declared as `public`

More on OOP

1. **Modifier**
   - **Access Modifier**: Private fields are accessible to all methods within the same class, regardless of which instance is being accessed.
   - **Static field and methods**:
     - A **static field** should be something that is **shared among all instances** of that class.
     - A **static method** performs a task which **doesn't need to access the data from any instance**.
     - In **static methods**, we **cannot access any instance fields or call other instance methods**, the keyword `this` is also **not allowed**!
     - **Static fields** can be **accessed** from **non-static methods.**
   - **`final`**:
     - `final` in a **field declaration** prevents **re-assignment**, in a **class declaration** prevents **inheritance**, in a **method declaration** prevents **overriding**.
   - **Modifier Order**: an example is `public static final void`, this is to declare a **constant**
2. **Heap and Stack**
   - **Stack**: The stack is where all variables (**primitive types and object references**) are stored. The stack contains **stack frames** (should be drawn as rectangles), stack frames are created when an **instance method** is called or when `main` is called, and it contains (**From bottom to up**)
     - the `this` reference
     - the method arguments
     - local variables within the method.
   - **Heap**: Whenever you use the keyword `new`, the **reference variable** is stored on the **stack**, the object is stored on the **heap**. An object in the heap contains the following (**From up to bottom**):
     - **Class name**.
     - Instance fields and respective fields.
     - Captured values
3. **Inheritance**:
   - The constructor of the subclass **should** invoke the constructor of the superclass via `super()`
   - Suppose we have two classes P and Q, if Q inherits from P, then we can say Q is the **subtype** of P or Q $<:$ P.
   - A class **cannot inherit from many classes**. (If it does, will generate **syntax error**, which belongs to **compile error**).
4. **Override vs. Overload**
   - **Override**: must have **same method descriptor (method signature + method return type)**, e.g.`A C::foo(B1,B2)`, (B1, B2 are the type of the method parameters, same for as follows)
   - **Overload**: must have same **method name**, in the same class and **different method signature (method name, number of parameters, type of each parameter, order of the parameters)**. e.g.`A C::foo(B1,B2)`. **The return type of the method doesn't matter.**
   - In the **subclass** of an **abstract class**, you still can **override the concrete method in that abstract class**.
   - Java **doesn't allow two methods with the same method descriptor existing in one class**. (Consider this with **bridge method**)!
   - If you override a method from the superclass, by conforming to LSP, when you call that function, it should invocate the overridden method.
5. **Tell, Don't Ask**: We never **ask** an object to spit out its own **raw data**. Instead, we **let the object know** what we want so that it can give us a piece of **processed data** (via an instance method).
   - **Sample reason during midterm**: The subclass should ask the super class to do the thing (to be changed).
6. **Liskov Substitution Principle**: A *subclass* **should not** break the expectations / **specifications** set by the *superclass*. a.k.a, the test cases that are passed in superclass should also be passed in the subclass. **Tips**:
   - Always write down what the specifications are set by the superclass.
   - Construct a method and test whether the subclass can be substituted without breaking the specifications. (If class B **extends** A, and **overrides** the method in A, then **successful substitution** means **when substitute A with B**, we should call the **overridden function in B**!)
7. **Method Invocation**: Pay attention to the **CTT, RTT** of the **target** and the **CTT** of the **parameter**. The **RTT** of the **parameter** doesn't matter!
   - During the compile time, find the **most specific** method descriptor. (Method M is **more specific than** method N means that the **type of the parameter of M** is the **subtype** of the **type of the parameter in N**.
   - During the run time, use the method descriptor we got from above to find **the first** method from the `RTT` to `Object` and execute it.
   - **Class Method invocation**: Only the first step will be taken and once the method is found, it will be executed.
   - **Type casting** happens during the **compile tile**! e.g. `(Circle) o2;`, the CTT of the method parameter is **Circle** even if `o2` might be an `Object`.
   - For **generic classes**,
     - the dynamic binding process generates the type-erased method signature **at the end of the compilation**, meaning that we look for the original class to find the method descriptor.
     - We store the **erased version** of the method descriptor.
     - During run-time, the dynamic binding process **only match the type-erased methods**, but **may include the bridge method**.
     - Build the erased version of the descriptor table

Class	Method Descriptor (Compile Time)	Remarks	Method Descriptor (Erased)	Remarks
A	void fun(T)	T <: Comparable<T>	void fun(Comparable)	-
A	<T> void fun(T)		void fun(Object)	-
B	void fun(Double)	Inherited from A<Double>	void fun(Comparable)	Bridge method

     - Build the method descriptor searching table

CTT (target)	CTT (param)	Accessible	Compatible	Most Specific
A	Integer	void fun(T) <T> void fun(T)	void fun(T) <T> void fun(T)	void fun(T)

     The accessible and compatible are the **method descriptors**, It is also recommended to add two columns, **erased method descriptor** and the **method invocated**
8. **Abstract class**: An abstract class in Java is a class that has been made into something so general that it **cannot be instantiated**! And it **can** have the following:
   - **Abstract method**: An abstract method **should not have** any method body but it **may throw an exception**! An abstract class without an abstract method is also allowed!
   - **Concrete method**: As the name suggests, methods that are **not** abstract are concrete!
   - **Instance/Class Field**: fields with `static` or without.
9. **Concrete Class**:
   - a concrete class must have **implementations for all inherited abstract methods** (if it extends an abstract class or implementst an interface). Otherwise, a **compile error**!
   - Beyond that, it's free to have whatever you want — or even nothing at all in terms of fields or methods.
10. **Interface**:
    - Interface **cannot** have **fields** and **concrete methods**!
    - If `C implements I`, then we have **C $<:$ I**. **A class can implement multiple interfaces.**
    - Given a class A and an interface I, even if we didn't specify `A implements I`, the code `I i = (I) new A();` still compiles because we are not sure **whether there is a subclass of A that implements I**. **If so, no compile and runtime error will be generated. If not, no compile error will be generated but a runtime error will be generated.**
    - **Always pay attention to the subtype realtionship containing interface**.
11. **`Object::equals`**: It will compare whether two objects are referenced to the **same memory address** or not. **Note**: To override this function from `Object` so it behaves as we want, we need to
    - check the RTT of `obj` is a **subtype** of the type we are interested (can be generic type), by using `if (obj instanceOf TYPE)`, if the TYPE is a generic type, it **must be an unbounded generic type, e.g. `A<?>`, it cannot be `A<String>`**
    - typecast `obj` to the type we are interested by using either the class name or generic type with unbounded wildcard, e.g. `Box<?>`, **always be careful when you want to type cast to a generic type, since you are casting it to a rawtype!**
    - Always pay attention to whether the `Object::equals(Object)` **has been overridden or not**. **If not, it will always compare whether two instances are the same or not**!

Exception & Wrapper Class

1. **Wrapper class**
   - **Auto-boxing**, e.g. `Integer i = 4`, only happens on **primitive type**. **Complex type**, like Java array, doesn't support auto-boxing. e.g. `int[]` won't be converted to `Integer[]` automatically.
   - **Unboxing also happens automatically**, it converts an instance of a wrapper class to its primitive type.

- Wrapper class objects are **immutable**, meaning that once you instantiate, changing the value will result in creating a new instance.

2. **Variance Relationship**: Let S denote the type of element in the "array". Then the **complex type** have three possible variance relationship:
   - **Covariant**: if `S <: T`, then `C(S)<:C(T)`. e.g.e **Java array `Integer[] <: Double[]` int[] < / : double[]**
   - **Contravariant**: if `S <: T`, then `C(T) <: C(S)`
   - **Invariant**: it is neither **covariant** nor **contravariant**.
3. **Application of CTT and RTT**
   - (**CTT**): To see whether a code will generate compile-error or not, we only see the CTT of the variable and the **type casting**.
   - (**RTT**): **Run-time** error judgment **only** needs us to see the **RTT** of the variable. We **must ignore** the type casting because Java is **strongly typed**, meaning objects always retain their actual type (RTT).
   - `(C) new B()` means the CTT is first B and then explicitly casted to C.
4. **Exception**: **Exceptions always happen at runtime!**
   - **Unchecked Exception**: It is a subclass of `RuntimeException`, which is a subclass of `Exception`. **Not necessary to be handled** but it is recommended to do so. If not, **compile error** will be generated!
   - **Checked Exception**: It is a subclass of `Exception`. **Must be handled**.
   - **Handle the exception**:
     - This **must be done** in the **`catch`** block or be passed to another "catch" block. If there is no need for the `finally` block, can omit it.
     - In the catch block, if there are blocks that are **unreachable**, a **compile error will be generated**!
   - If we have the code `int[] arr = new int[3]; arr[5]=10`, it **won't generate compile error, but will generate runtime error**!
   - When an inner function throws a **checked exception**, if its outer function didn't handle it, it will "pass" the exception all the way until it is caught (**throw** immediately).
   - The `finally` block is **always executed** even when `return` or `throw` is called in a `catch` block. (`throw` can be interpreted as `return` for easy understanding)
   - It is possible for an Overriden method to throw the **same exception** or **any of its subtypes** as the method in the **parent class**. **Throwing a supertype of the parent class's exception is not allowed**!

Generics & Wildcards

1. **Generic Type**:
   - **Constructor**: the constructor of a generic type shouldn't contain <> operator. **Note**: when we **call** the constructor, we **must include <> operator**
   - **Factory method**: it is a **class** method (declared with `static`) and a **generic** method (declare a method-level type parameter). e.g., `public static <T> Box<T> of(T obj) { return new Box<T>(obj); }`. **Factory method is not a constructor!**
   - **Parameterize a generic type**:
     - When we use `extends` or `implements` a generic type `T`, we **must** instantiate the generic type `T`!
     - When we call a method from a generic type, we should also **parameterize** the generic type either explicitly, e.g. `Box<String>`, or implicitly, e.g. `Box<>` (**must include <>**)
     - **Rule of Thumb**: Always think about **which generic type is the one you want to instantiate**!
   - **Subtype between generic type**: If you explicitly use `extends/implements`, e.g. `class A<T> extends B<T>`, then `A<T>` is a **subtype** of `B<T>`.
   - **Bounded generic type parameter**: `T extends Class & Interface`, **the first bound must be a class**! Otherwise, compile error will be generated!
   - **Generics are invariant**: If `S<:T`, **`A<S> </: A<T>`**!
2. **Generic method**:
   - **Method-Level Type parameter**: When a method declares its own generic type parameter with the **same name** as the class-level type parameter, the method-level type parameter will **shadow** the class-level type parameter within the method's scope. And **these two parameters are not the same**!
   - **Class-level type parameter cannot be used in static method or static field**!
   - **Non-static Generic method**: e.g. `public <U> Box<U> map {}, public Box<S> map {}`, this kind of method **may or may not** declare **method-level** type parameter, it can use **class-level** type parameter. And it depends on design requirements.
     - **Invoke**: To invoke, we can use `instance.method()`
     - **Non-static generic method cannot be parameterized using <>**! Otherwise, a **compile error** will be generated!
   - **Static Generic method**: e.g. `public static <T> Box<T> ofNullable(T obj) { }`, this kind of method **must be declared using a method-level type parameter**.
     - **Invoke**: To invoke, we can use `ClassName.<Type>method()`, or we can **omit** the <Type> to let the compiler do the type inference.
   - **Field-level type parameter**: Java **doesn't have** field-level type parameter!
3. **Type erasure**
   - Replace **generic type** with its **raw type**.
   - Replace type parameters.
     - **Non-bounded type parameters** are replaced with `Object`
     - **Bounded type parameters** are replaced with **the first bound** and **explicitly cast to the second bound**.
   - Insert necessary cast (Usually narrowing conversion) to make sure casting to the expected type.
   - **Example**: this code `<U extends Container> void check(U con) {}` will become `void check(Container con) {}` **after type erasure**.
4. **Raw Type**
   - The **Type erasure** of the raw type doesn't have the last **casting step**. The remaining is the same.
   - When you use **raw type** in your code, there will always be a **rawtype warning**. (**Note that using unbounded wildcard <?> to replace rawtype won't generate a rawytype warning!**
   - If rawtype is used, inside the generic type, all the type parameter will become `Object` or the **first bound**! Remember this!
   - **Classic Example**

```
class Store<T> {
  T x;
  void keep(T x) {
    this.x = x;
  }
  T get() {
    return this.x;
  }
}
```

```
Store<String> stringStore = new Store<>();
Store store = stringStore;

store.keep(123);  // Line A

String s = stringStore.get(); // Line B
```

   During compile time, `T` in `store` is `Object`, thus `Integer` is allowed!. During run-time, type erasure erased all `T` to `Object`. Thus, Line A is also allowed, but since Line B involves explicit casting between `String` and `Object`, this is not allowed!
5. **Generic Array**:
   - We **cannot instantiate** a Java array using the type parameter, e.g. `new T[]` is not allowed. However, we **can declare** a Java array using the type parameter, e.g. `T[]` a is allowed.
   - **An example**: `@SuppressWarnings("unchecked")`, then `Queue<Passenger>[] temp = (Queue<Passenger>[]) new Queue<?>[totalStops]`
   - The generic array you declared after using the above method is **nothing but a Java Array**, it has `length` property!
6. **Bridge method**: A bridge method is **always generated** when, 1) a type **extends/implements a parameterized type** and 2)type erasure **changes the signature** of **one or more inherited method**, which makes direct overriding impossible.

```
class A<T> {
  public void fun(T x) {
    System.out.println("A");
  }
class B extends A<String> {
  public void fun(String i) {
    System.out.println("B");
  }
}
```

After Type Erasure

```
class A {
  public void fun(Object o) {
    System.out.println("A");
  }
}
class B extends A {
  public void fun(Object o) { // Bridge method
    this.fun((String) o);
  }

  public void fun(String i) {
    System.out.println("B");
  }
}
```

   - **Steps**:
     - copy the erased method into the subclass.
     - Insert the cast using the **type argument** and call the available method using `this` (If **can** find a method matching the argument type, a.k.a subclass overrides the superclass) or `super` (If **cannot** find such a method, a.k.a no overriding).
   - **Calling bridge method**: With the above example, the following code will call the bridge method `A<String> a = new B(); a.fun("2");` and it will print "B". Notice when you use `A<String>`, the type parameter in class A will be **indicated** as `String`, treat `T` as `String` when **finding the method descriptor**, but when storing the method descriptor, still treat it as `Object`.
   - Bridge methods only appear when overriding a **method** that uses the **class's type parameter**.
7. **PECS Rule**: Producer `extends`, consumer `super`. **Note that PECS is usually used on method parameter.** An easy way to think of it is as follows
   - Take the method parameter as your `studyObject`
   - look at the `studyObject.method()`
   - If `.method()` is something like `get(), read()`, then your `studyObject` is a producer, add **lower-bounded wildcard** to your method parameter.
   - If `.method()` is something like `set(), write()`, then your `studyObject` is a consumer, add **upper-bounded wildcard** to your method parameter.
8. **Wildcards**
   - Wildcards **is not a type!**, so, you **cannot** use them in class declaration and **cannot use them as type arguments**! But wildcards **can be used to instantiate an array of generic types**.
   - **The following is not allowed**
     - `private static final Box<?> emptyBox = new Box<?>(null);`
     - `public <T> of(<? extends T>[], int depth)`
   - **Upper-Bounded Wildcards**: A<? extends T>, an upper-bounded wildcard allows a generic type to accept any **subtype** of a specified class or interface `T`.
     - If `S<:T`, then A<? extends S><: A<? extends T>**(Covariance)** It will be beneficial to use **subtype is nothing but subset** to understand this relationship!
     - For any type S, A<S><: A<? extends S>
   - **Lower-Bounded Wildcards**: A<? super T>, a lower-bounded wildcard allows a generic type to accept any **supertype** of a specified class or interface `T`.
     - If `S<:T`, then A<? super T><:A<? super S>**(Contravariance)**.
     - For any type S, A<S> <: A<? super S>
   - **Unbounded Wildcards**: `A<?>`
     - `A<?>` is the **supertype** of every **parameterized type** of `A<T>`, that is `A<T><:A<?>`.
   - During **Type erasure**, wildcards will be erased! And generics become the **raw type**!
9. **Type inference**
   - **Rule to find constraints**
     - **Target**: "the **return type** of the method" ¡: "the type of the variable you are assigning to"
     - **Argument**: "the type of the **argument**" ¡: "the type of the **parameter**"
     - **Bound**: we need to consider "the **bound of the generic type parameters**"
   - **Rules to solve constraints**
     - `Type1<:T<:Type2`, then `T` is inferred as `Type1`
     - `Type1<:T`, then `T` is inferred as `Type1`
     - `T<:Type2`, then `T` is inferred as `Type2`
   - **Type inference involves wildcard**
     - If parameter type is `Seq<? super T>`, argument type is `Seq<G>`, then `T<:G`
     - If parameter type is `Seq<? extends T>`, argument type is `Seq<G>`, then `G<:T`
   - If `class A implements Comparable<A>`, and `class B extends A`, then `B` actually implements `Comparable<A>` **not** `Comparable<B>`!